*Journal of Software Engineering Research and Development, 2024, 12:8,* doi: 10.5753/jserd.2024.3591

# Technical Debt Tools: a Survey and an Empirical Evaluation

**Tchalisson Brenne S. Gomes** [ **State University of Piauí – UESPI** | ***tchalisantos40@gmail.com*** ]
**Diogo Alves de Moura Loiola** [ **State University of Piauí – UESPI** | ***diogoloilola@gmail.com*** ]
**Alcemir Rodrigues Santos** [ **State University of Piauí – UESPI** | ***alcemir@prp.uespi.br*** ]

*Background:* The life cycle of a technical debt from its identification to its payment is long and may include several activities, such as identification and management. There is a lot of research in the literature to address different sets of these activities by different means. Specifically, several tools have already tackled such technical debt identification problems. However, only a few studies empirically assessed those tools. *Method:* In this article, we carried a multi-method research. We first surveyed the literature for the technical debt tools available and then we evaluated two of them, which aim at identification of self-admitted technical debt. They are named eXcomment e DebtHunter. *Results:* We found 97 tools employing different approaches to support technical debt life cycle management. Most of them (59%) address only the high level task of management, instead of actually identify and pay the debt. Additionally, as for our empirical evaluation of tools, our results show that DebtHunter found only 7% of debt identified by eXcomment. In the other way around, eXcomment found 19.9% the debt found by DebtHunter. Besides, both tools have low levels of *precision* and *recall*. *Conclusion:* It is hard to find technical debt through comments. Both tools can find indicators of debt items, however they struggle on the precision and recall. In fact, although eXcomment and DebtHunter diverge on the amount of debt identified, they seem to converge with regard to the type of debt present in the system under evaluation.

***Keywords:*** *Empirical Evaluation; Survey; Technical debt; Tools;*

## 1 Introduction

Technical debt (TD) is a metaphor introduced by Cunningham (1992) which indicates the shortcuts taken during the software development activity that may affect the software quality. Such shortcuts incur extra costs in the future in the form of increased cost of change during software evolution and maintenance. Technical debt items can occur in different artifacts throughout the life cycle of a software project. The presence of instances of technical debt can occur in two distinct ways: *i)* `non-admitted` – among other reasons[1], due to low quality code due to the lack of experience developers; or *ii)* `self-admitted` (SATD) – when developers introduce low quality code in the project deliberately aiming at prioritizing other tasks and report it in the comments or other artifacts.

In fact, technical debt items are certainly one of the most investigated problems by the software engineering community in recent years. However, the management of technical debt still represents a major challenge in the community. For instance, Rios et al. (2018) presented a tertiary study on the topic, highlighting the types of technical debt, as well as the point in the software life cycle they can occur along with the management strategies present in the literature. In addition, they provided a thorough overview of technical debt and its implications for professional practice.

Although the presence of technical debt in a project might be inevitable (Tom et al., 2013), preventive actions can be taken so that debts do not get out of control, such as creating test cases, clear definition of requirements, and good project planning (Freire et al., 2020). One of the most used ways to tackle this problem is the adoption of automated approaches that can support these tasks (Avgeriou et al., 2021). Among them, tools employing approaches based on static code analysis and source code comment mining stand out (Avgeriou et al., 2021). However, regarding the high amount of tools available in the literature (regardless of its license – either commercial or research prototypes), Avgeriou et al. (2021) argue this can become a problem for developers who wish to select any of them for daily use. It is necessary to evaluate and compare existing tools to ensure the effectiveness of these tools, as well as to facilitate the developers' choice process, in addition to assisting in their evolution process.

In this sense, several studies from the literature have tried to evaluate such tools from different perspectives. Li et al. (2023) proposed the identification of self-admitted technical debt from four different sources: source-code comments, issue tracking systems, pull requests, and confirmation messages. Avgeriou et al. (2021) used a qualitative group evaluation using three main criteria: characteristics, popularity, and empirical validation. The results indicate that only SonarQube, DV8, and SonarGraph have been considered in empirical studies regarding the TD index and that more studies are needed to be able to quantify debt repayment efforts. In addition, although Fowler (2018) argues that *code smells* are not problematic on their own, Gomes et al. (2019) highlighted the relationship between *code smell* and self-admitted technical debt. They used automatic detection of *code smells* to identify debts present in the code, since they are considered indicators of the presence of technical debt.

However, when it comes to extraction of information from software comments by mining repositories, the evaluation and comparison of TD identification tools is not trivial (Farias et al., 2021; Loiola et al., 2023). It requires a robust data set which must have unambiguous classification of the classified debt items. In addition, a distinct data set

[1]The Technical Debt Quadrant, available at Fowler's web page (`https://martinfowler.com/bliki/TechnicalDebtQuadrant.html`).

is required for each automatic technical debt identification approach. In a first attempt to build a self-admitted technical debt items data set, Maldonado and Shihab (2015) used of source code comments in the identification of self-admitted technical debt showing promising results. Software comments represent a valuable source of information, once they can describe the programming tactics used in a given portion of the code. Besides, they can report bad coding practices or divergence from the accorded requirements. In fact, Maldonado and Shihab (2015) manually identified and quantified different types of self-admitted technical debt in open-source systems. Their results showed the most common types of debt that can be found in the comments are related to *(i)* design, *(ii)* defect, *(iii)* documentation, *(iv)* requirements, and *(v)* test, respectively.

In this context, Oliveira et al. (2020) evaluated two tools: SonarQube (which employs static code analysis) and SATD-Detector (which employs source code comment mining). The study identified the intersection between both approaches. As results, it was found that only $19.47\%$ of the debt items were identified by both evaluated tools. In another study, Loiola et al. (2023) evaluated the effect of the use of context heuristics in the self-admitted technical debt mining in software comments. They found the heuristics reduce the amount of false positives, yet they were not enough to make the evaluated tool to reach good levels of *precision*.

In our previous work (Gomes et al., 2022), we investigated the intersection of results from two self-admitted technical debt identification tools by mining source code comments. In this paper, we extend our previous work by both *(i)* including a literature survey we performed prior to the empirical study on the existing technical debt identification tools and *(ii)* investigating a third research question regarding the effectiveness of the tools. For the sake of completeness, the results and discussions of the preliminary work have been replicated in this paper. The results from the original study show that out of the debt items classified by eXcomment, only 7% of them DebtHunter also classifies as debt and that, of those items that DebtHunter classified as debt, only 19.9% of them eXcomment also classified. In fact, despite differing in the amount of technical debt items, the tools seem to converge regarding the types of debt present in the systems evaluated. In addition, the research question addressed in this extension show that both tools have low levels of *precision* and *recall* in the task of classifying the debt items they manage to identify, and that they diverge when they incur into false positives or false negatives.

In summary, this paper presents the following contributions:

(i) A bibliographic survey of the technical debt identification tools available in the literature;
(ii) An empirical evaluation of two self-admitted technical debt identification tools (eXcomment and DebtHunter);
(iii) The construction of an oracle with technical debt from four open-source systems[2];
(iv) A discussion of the errors of the evaluated tools to serve as a basis for improvement for tools developers.

[2]Available at Zenodo Gomes et al. (2024)

The rest of the paper is organized as follows. Section 2 presents the work we deemed as related to ours. Section 3 presents the study planning, while the Section 4 presents our survey of tools available in the literature and the Section 5 presents our empirical evaluation of two technical debt tools, followed by Section 6 which presents the threats to validity. Finally, Section 7 presents our concluding remarks, as well as the future work.

## 2 Related Work

This section presents some research we deemed as related to ours (Li et al., 2015; Bavota and Russo, 2016; Gomes et al., 2019; Farias et al., 2020; Oliveira et al., 2020; Lenarduzzi et al., 2022, 2021a). Next, we discuss them separately, first the literature reviews, then the empirical studies.

### 2.1 Other Reviews

We identified other technical debt literature reviews already published in the literature (Li et al., 2015; Alves et al., 2016; Rios et al., 2018; Lenarduzzi et al., 2022; Murillo et al., 2023). Out of them, only Avgeriou et al. (2021) focused on the available debt management tooling. However, Li et al. (2015); Alves et al. (2016); Rios et al. (2018); Lenarduzzi et al. (2022); Murillo et al. (2023) also dedicated some attention to the topic as they address the identification of technical debt items. Table 1 summarizes other reviews contributions.

Li et al. (2015) analyzed the technical debt concept on 94 existing research efforts and they proposed a classification of ten technical debt types. They identified the quality attributes compromised by technical debt and determined activities and tools for technical debt management. Alves et al. (2016) carried a mapping study on the identification and management of technical debt items, which was recently updated by Murillo et al. (2023). In the latter study, they studied the evolution of the area in comparison with the former. They found that researchers are more likely to investigate new technical debt as general problem instead of an isolated problem, in addition, they assess their new approach rather than compare with available baselines. Rios et al. (2018) carried a tertiary study on the current state of research on technical debt in general. To what is worth, they explored what types of approaches and tools are being proposed to assist in monitoring technical debt superficially. Their result highlighted the poorly explored points in the technical debt scenario.

Recently, Avgeriou et al. (2021) provided an overview of the current landscape of technical debt measurement tools, comparing features and the popularity of tools, as well as the analysis of existing empirical assessments. Their review allowed the comparison of several nine tools in the literature aiming at assisting developers in the choosing of a tool that meets their needs. Although, the scope of their study was limited to three most common types of technical debt, namely *code*, *design*, and *architecture*, Murillo et al. (2023) fund that most of the research carried on the identification of the technical debt items since 2015 address these kind of debt.

Murillo et al. (2023) study itself updated the mapping study of Alves et al. (2016) on the identification and man-

**Table 1.** Contributions by other literature reviews.

| Reference | Research Topic | Analyzed Period | Contributions |
|---|---|---|---|
| Li et al. (2015) | Technical debt management | 1992 –2013 | Determined activities and tools for technical debt management. |
| Alves et al. (2016) | Technical debt identification and management | 2010 -2014 | Listed strategies to identify or manage technical debt |
| Rios et al. (2018) | Technical debt | 2012-2018 | |
| Lenarduzzi et al. (2021a) | Technical debt prioritization | 2011-2020 | Identified tools for technical debt prioritization. |
| Murillo et al. (2023) | Technical debt identification and management | 2015-2022 | An update of Alves et al. (2016). |

agement of technical debt. With regarding the identification of debt items, they were interested in the empirical studies, artifacts, data sources, visualization techniques proposed in the analyzed period. Additionally, they also discussed the empirical evaluations and software visualization techniques newly available in the period. Although they did mention some tools available for these purposes, they do not discuss their details in depth.

## 2.2 Empirical Studies

Bavota and Russo (2016) replicated the work of Potdar and Shihab (2014) aiming at identifying the propagation and evolution of self-admitted technical debt. For this purpose, more than 600,000 *commits* and two billion comments were collected. The results of the study showed that self-admitted technical debt is diffuse, with an average of 51 instances per system. It was also found that it is mainly composed of code debt and requirements, and that debts increase over time due to the introduction of new instances.

Gomes et al. (2019) investigated the relationship between code smells and self-admitted technical debt. They selected three *open-source* projects written in `Java`: ArgoUML, JFreeChart and Apache Ant. Their results show there was a strong correlation between self-admitted technical debt and code smells. Besides, in some cases, the use of source code comments can complement information that could not be obtained with the use of code smells alone.

Farias et al. (2020) carried out a study to identify TD through source code comments in the project using their automated approach. They used their tool, named eXcomment, which is able to mine the source code comments and it uses natural language processing techniques along with a contextualized vocabulary they developed. They managed to identify several types of debt in the comments of the systems evaluated.

In a similar initiative, Oliveira et al. (2020) conducted a preliminary study to compare two different approaches to identifying TD. The first one, the SonarQube tool and the second one the SATDDetector. Their results indicate there is a certain amount of debt that can be identified by both approaches, but the authors state that there is still a lack of studies for a greater accuracy of what these results really mean.

Lenarduzzi et al. (2021a) performed a literature review regarding strategies and tools for technical debt prioritization. Code, architecture, and design were the most frequent types of technical debt addressed. Afterwards, Lenarduzzi et al. (2022) performed a analysis of six static analysis tools that among other things, they aim to control quality issues like technical debt: Better Code Hub, CheckStyle, Coverity Scan, FindBugs, PMD, and SonarQube. Their key results show little to no agreement among the tools and a low degree of precision aligning with with our results.

Comparing to the other studies, our study is the only one to compare eXcomment and DebtHunter tools against each other with regarding their agreement and effectiveness. It is worth mention that, while evaluating such tools, we are comparing two different approaches to identify technical debt items through the source code comments mining.

## 3 Study Setup

We performed this study in two phases: a literature review and an empirical study. Figure 1 shows the study planning. The literature review serves the purpose of identifying the available tools for technical debt items identification. We discuss the results of the literature review in the Section 4. Out of available tools, we selected two of them to perform the empirical study, which we discuss in the Section 5.

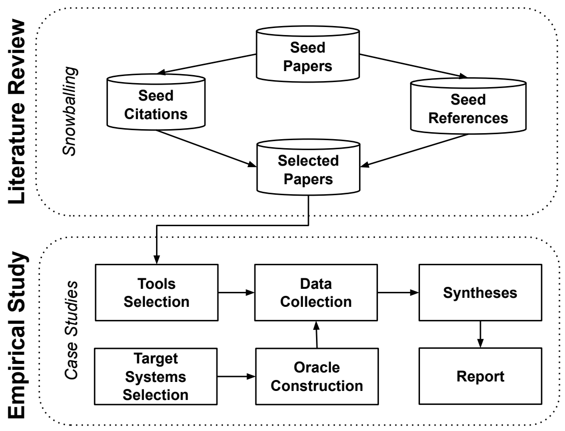


**Figure 1.** Study execution phases.

## 4 Survey of Technical Debt Tools

We reviewed the literature seeking to answer the following question:

**RQ$_1$:** What are the tools supporting the management of technical debt items available in the literature?

More specifically, we want details on the purpose, license, languages supported, type of debt addressed, and evaluation. Thus, we specified the main question into sub-questions as follows.

RQ$_{1a}$: What are the purposes employed by the tools available in the literature in technical debt management?
RQ$_{1b}$: What are the licenses for use and availability of the tools available in the literature?
RQ$_{1c}$: What programming languages or *frameworks* do the tools available in the literature support?
RQ$_{1d}$: Which tools propose to identify non-admitted or self-admitted technical debt?

Next, we detail the selection process and results.

### 4.1 Studies Selection

We used the *snowballing* method for the literature review, in which we start from some known articles and proceed by following references back and forth from them. That is, to identify articles cited by the known article and to identify articles that cite the known article. We used a set of articles as the basis for *snowballing* (Avgeriou et al., 2021). We took them as a starting point because they carried out an in-depth review of the literature regarding technical debt and specifically on tools and identification of for different aspects of the debt items life cycle.

In order to get to the final set of papers for this review, we enumerated all references (R) and citations (C) – by the end of 2022, in the page of the publisher – of the base article Avgeriou et al. (2021). First of all, we attempted to retrieved all the papers, 44 in total (12 R + 32 C). Unfortunately, four citation papers we did not manage to overcome the paywall (C1, C7, C11, and C13). The citation C10 is the paper itself, which is still mistakenly accounted as a citation in publisher page by the time of writing (end of 2023). Thus, 5 papers were filtered. Then, we performed the scanning and skimming of the collected papers and filtered those that we did not find any tool of interest. At the end, besides the base paper, thirteen papers were selected for a full reading and data extraction. Table 2 summarizes this filtering process followed until we reach the final set of papers, which are enumerated in the Table 3. The other references are available in the replication package Gomes et al. (2024).

### 4.2 Data Extraction

For data extraction, the selected articles were read in full after the selection steps discussed in Section 4.1. In this step, we sought to extract information about the tools that support technical debt management. In addition, relevant information about them was extracted, such as source code license (*i.e.* the tool has open source code), as well as its availability (*i.e.* the tool was found available to be used), in addition to the programming language it supports. It was also possible to extract information about the purpose employed by the tools regarding the context of the debt life cycle.

**Table 2.** Article selection steps to review the existing technical debt tools in the literature.

| Step | References | Citations | Total |
|---|---|---|---|
| Base Article | R1 – R12 | C1 – C32 | $1 + 44$ |
| Filter 1 | R1 – R12 | C2–C6, C8, C9, C14, C15 – C32 | $1 + 39$ |
| Filter 2 | R2, R6, R7, R9 | C9, C14, C16, C18, C23, C24, C25, C27, C31 | $1 + 13$ |

**R:** Reference; **C:** Citation.

### 4.3 Review Results

Throughout this review, we found 101 tools employing different approaches tho support debt management. As an example, Avgeriou et al. (2021) that aimed at provide an overview of the current landscape of TD measurement, only considered 26 and analyzed only 9 tools and the study of Rios et al. (2018) mentioned only 41. We now present such tools from the perspective of criteria established by our research questions. Tables 4, 5, and 6 present such tools and follow the same structure. The column "Tool" presents the name of the identified tools. The column "O" indicates which tools are licensed as open-source. The column "A" indicates which tools available for use. The column "L" indicates which programming languages the tools support. Finally, the column "P" indicates the purpose of the tool in relation to the context of the debt life cycle, which can be: *(i)* identification, *(ii)* management, *(iii)* testing or *(iv)* visualization.

#### 4.3.1 RQ$_{1a}$: on the purpose of the tools

***Identification, Visualization, or Testing***: Table 4 shows the results regarding the purpose of identification, visualization or testing. Considering the identification, we found only $8\%$ $(9/101)$ of the tools propose exclusively to identify the existing debts in the project, and most of them use approaches based on static code analysis for the identification of debts. Regarding the visualization of technical debt, it was possible to identify that only $2\%$ $(3/101)$ of the tools employ purposes related to the visualization of debts in the project, indicating that this type of approach is still little explored by the tools. It was also observed that test case management tools are still little explored. They represented only $3\%$ $(4/101)$ of the total. Test case management tools provide support to the development of test cases. Somehow, they help developers to avoid test debt. For instance, the "Parasoft Jtest" tool that optimizes *JUnit* test cases and while it improves the test coverage of the project.

***Management***: It was observed that $58\%$ $(59/101)$ of the tools have the purpose of management, *i.e.* they do not address any specific type of technical debt. In fact, they focus

**Table 3.** Selected papers for review.

| ID | Title | Reference |
|---|---|---|
| R2 | Technical Debt Indexes Provided by Tools: A Preliminary Discussion | Fontana et al. (2016) |
| R6 | A survey on code analysis tools for software maintenance prediction | Lenarduzzi et al. (2020) |
| R7 | A systematic mapping study on technical debt and its management | Li et al. (2015) |
| R9 | A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners | Rios et al. (2018) |
| C9 | On the Lack of Consensus Among Technical Debt Detection Tools | Lefever et al. (2021) |
| C14 | A critical comparison on six static analysis tools: Detection, agreement, and precision | Lenarduzzi et al. (2022) |
| C16 | SDK4ED: A platform for technical debt management | Ampatzoglou et al. (2022) |
| C18 | analyzeR: A SonarQube plugin for analyzing object-oriented R Packages | Chandramouli et al. (2022) |
| C23 | Exploring Technical Debt Tools: A Systematic Mapping Study | Freitas et al. (2022) |
| C24 | Customizable visualization of quality metrics for object-oriented variability implementations | Mortara et al. (2022) |
| C25 | Technical Debt in Service-Oriented Software Systems | Nikolaidis et al. (2022) |
| C27 | TD Classifier: Automatic Identification of Java Classes with High Technical Debt | Tsoukalas et al. (2022) |
| C31 | A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools | Lenarduzzi et al. (2021b) |

**Table 4.** Identification, Visualization or Testing tools.

| Tool | O | A | L | P |
|---|---|---|---|---|
| CAST | ☐ | ☑ | Many | I |
| RE-KOMBINE | ☐ | ☐ | Not identified | I |
| Sonar TD plugin | ☑ | ☑ | Java | I |
| JSpirit | ☑ | ☑ | Java | I |
| MAT | ☐ | ☐ | Java | I |
| SATD Detector | ☑ | ☑ | Java | I |
| TD-Tracker Tool | ☐ | ☐ | Java | I |
| DebtHunter | ☑ | ☑ | Java | I |
| Code Christimas Trees | ☐ | ☐ | Not identified | V |
| Structure101 | ☐ | ☑ | Java, .NET | V |
| ParasoftJtest | ☐ | ☐ | Java. C, C++, .NET | T |
| IBM appScan | ☐ | ☑ | Java | T |
| ParasoftdotTEST | ☐ | ☐ | Java, C/C++, .NET | T |
| Titan | ☐ | ☐ | Java | T |
| VariMetrics | ☑ | ☑ | Java | V |

**O:** Open-source; **A:** Available; **L:** Language(s) supported; **P:** Purpose;
**I:** Identification; **V:** Visualization; **T:** Testing;

on tracking and monitoring it. As in the tool CheckStyle, which supports writing code following a coding standard, which can be an important factor for controlling debt in a software project. Another tool, FxCopAnalyzer can be integrated with the Visual Studio IDE or used in a standalone mode. It can provide visual and interactive feedback to developers, making it easier the identification and correction of problems related to the code quality. Thus, such tools support developers to write code that conforms with predefined coding rules. In other words, such functionalities help developers to avoid code debt.

In addition, tools that use static code analysis to identify deep problems related to software quality stand out, such as *architecture-smells* and *code-smells*. As in the tool inFusion, which uses static code analysis to detect *code-smells* and *architecture-smells* present in the project and in the CAF-FEA, which allows to identify and prioritize problems related to architecture debts, such as the lack of modularization, inappropriate use of design patterns or components, besides the lack of documentation. These tools use a consolidated set of metrics related to code quality assessment, which allows a more comprehensive and reliable evaluation of the developed code.

***Multi-purpose or Unidentified*:** $19\%$ $(20/101)$ of the tools are multi-purpose with regard to the technical debt life cycle. For instance, SonarQube in addition to performing the identification of debts, it has functionalities related to the management and visualization of technical debt in the project. Another example is the *Understand* tool that allows the code analysis through visualization metrics in its integrated development environment. This tool support developers in the comprehension, maintenance, and documentation of source code.

Finally, we could not identify the purpose of $5\%$ $(6/101)$ tools. This indicates that, although there are many tools cited in the literature, there is still a lack of information available about them regarding the construction and approach employed, which may make it difficult for researchers to evaluate existing tools or to continue the work carried out by previous research and development.

### 4.3.2 RQ$_{1b}$: licenses of use and availability

Only 29% $(30/101)$ of the identified tools are both open source and available for use. Nevertheless, although they are open source and available, *(i)* some tools are commercial tools, which means they may have paid functionalities or ser-

**Table 5.** Tools with purpose of management of technical debt.

| **Tool** | **O** | **A** | **L** |
|---|---|---|---|
| analyzeR | ☑ | ☑ | R |
| inFusion | ☐ | ☑ | Not identified |
| CodeSonar | ☑ | ☑ | Java, C, C++ |
| Polyspace | ☐ | ☑ | C, C++ |
| CheckStyle | ☑ | ☑ | Java |
| Klocwork | ☐ | ☐ | Java, C, C++, C# |
| JLint | ☐ | ☐ | Java |
| Lattix | ☐ | ☑ | Java, C, C++ |
| Fortify Satic Code Analyzer | ☐ | ☑ | Many |
| ConQAT | ☑ | ☑ | Java, C#, C++, ABAP, ADA |
| Ndepend | ☐ | ☑ | .NET |
| LDRA testbed | ☐ | ☑ | Java |
| Axivion Bauhaus Suite | ☐ | ☑ | Java, ADA, C/C++ e C# |
| Source meter | ☑ | ☑ | Java, C/C++, C#, Python e RPG |
| Imagix | ☐ | ☑ | Java, C, C++ |
| Codacy | ☐ | ☑ | JavaScript, Java, PHP, Python, Ruby, Swift, C/C++ |
| SIG Software Analysis Toolkit | ☐ | ☑ | Not identified |
| Google CodePro Analytix | ☐ | ☑ | Not identified |
| iPlasma | ☐ | ☐ | Java, C/C++ |
| Eclipse Metrics | ☑ | ☑ | Java |
| Rational AppScan | ☐ | ☑ | Java, .Net, Javascript |
| PHPMD | ☑ | ☑ | PHP |
| FxCop | ☐ | ☑ | .NET |
| CodeXpert | ☐ | ☐ | Not identified |
| CAST Software's Applications Intelligence Platform (AIP) | ☐ | ☑ | Java, .NET, C/C++, Fortran |
| STAN | ☐ | ☑ | Java |
| Better Code Hub | ☐ | ☑ | Many |
| Resource Standard Metrics | ☐ | ☑ | C/C++, C# e Java |
| RBML compliance checker | ☑ | ☑ | Java, .NET |
| A tool to identify bad dependencies | ☐ | ☐ | C/C++ |
| SonarQube COBOL Plugin | ☑ | ☑ | COBOL |
| CLIO | ☐ | ☐ | Java |
| CodeVizard | ☐ | ☐ | Java, C# |
| Designite | ☑ | ☑ | Java |
| CodeInspector | ☐ | ☐ | Many |
| CodeMRI | ☐ | ☑ | Java |
| DV8 | ☐ | ☐ | Not identified |
| SQuORE | ☐ | ☑ | Many |
| SymfonyInsight | ☐ | ☑ | Many |
| Archinaut | ☑ | ☑ | Java |
| CBRI-Calculation | ☑ | ☑ | Perl, Python |
| DBCritics | ☐ | ☐ | PL/SQL |
| TEDMA | ☐ | ☐ | Java, R |
| BPMNspector | ☑ | ☑ | Java |
| ARCAN tool | ☐ | ☑ | Java |
| SAApy | ☑ | ☑ | Python |
| Code Analysis | ☐ | ☑ | Many |
| FITTED | ☐ | ☐ | Java |
| JCaliper | ☑ | ☑ | Java |
| ProDebt | ☐ | ☐ | Java, .Net, e Objective-C |
| DebtGrep | ☐ | ☐ | C/C++ e assembly |
| Hansoft | ☐ | ☑ | Many |
| Jacoco | ☑ | ☑ | Java |
| Jira Software | ☐ | ☑ | Many |
| Redmine | ☑ | ☑ | Ruby |
| SonarCloud | ☐ | ☑ | Many |
| TeamScale | ☐ | ☑ | Java, C#, C/C++, JavaScript, e ABAP |
| Visual studio FxCop-Analyzer | ☐ | ☑ | .NET |
| Scitool understand | ☐ | ☑ | Many |

**O:** Open-source; **A:** Available; **L:** Language(s) supported.

vices, *e.g.* SonarQube; *(ii)* some are dormant, due to the lack of a community responsible for its maintenance, including the management of dependencies, which can lead unusable or hard to be evolved or even to be evaluated by other researchers. It is hard, if not impossible, to evaluate something you cannot use. We believe this is one of the main reasons behind low number of evaluation studies. Theses facts reinforce the call for attention on the sustainability of research software (Venters et al., 2018). In addition, 38% (39/101) are only available for use, but they do not have an open source license, such as the tool AnaconDebt that allows to evaluate the cost of refactoring and the interest incurred on technical debt, although it is available, it is not open source and the interest estimation approach on the debt is confidential. Moreover, we could not identify neither the license type nor the availability of 31% (32/101) of the tools. This fact might indicates the lack of interest on the continuation of research on the approaches used and their development.

**Table 6.** Tools with several purposes and tools did not identify its purpose.

| Tool | O | A | L | P |
|---|---|---|---|---|
| SonarGraph | ☐ | ☑ | Java, C, C++, C#, Python | I, G |
| SonarQube | ☑ | ☑ | Many | I, G, V |
| FindBugs | ☐ | ☑ | Java | G, T |
| Coverity | ☑ | ☑ | Java, C, C++ | G, T |
| PMD | ☑ | ☑ | Java | G, T |
| Jarchitect | ☐ | ☑ | Java | G, V |
| NCover | ☐ | ☑ | .NET | G, T |
| Cobertura | ☐ | ☐ | Not identified | G, T |
| Software Maps tool | ☐ | ☐ | Not identified | G, V |
| Technical Debt Evaluation (SQALE) plugin for SonarQube | ☐ | ☑ | Many | I, G, V |
| DebtFlag | ☐ | ☐ | Java | I, G |
| CodeScene | ☐ | ☑ | Many | G, V |
| AnaConDebt | ☐ | ☐ | Not identified | I, G |
| eXcomment | ☑ | ☑ | Java | G, I |
| Visminer TD | ☑ | ☑ | Java | I, G, V |
| DeepSource | ☐ | ☑ | Java | G, I |
| Themis | ☐ | ☐ | Java | G, I |
| Build Game | ☐ | ☐ | Not identified | NI |
| Georgios Tool | ☐ | ☐ | Not identified | NI |
| Requirements Specification Tool | ☐ | ☐ | Not identified | NI |
| TD tool | ☐ | ☐ | Not identified | NI |
| Dependency Tool | ☐ | ☐ | Not identified | NI |
| MIND | ☐ | ☐ | Not identified | NI |
| CAFFEA | ☐ | ☐ | Not identified | I, G |
| SDK4ED | ☑ | ☑ | Java, C, C++ | I, G, V |
| SmartCLIDE Eclipse Theia Extension | ☑ | ☑ | Many | I |
| TD Classifier | ☑ | ☐ | Java | I, V |

**I:** Identification; **V:** Visualization; **G:** Management; **NI:** Unidentified. **O:** Open-source; **A:** Available; **L:** Language(s) supported. **P:** Purpose;

#### 4.3.3 RQ$_{1c}$: supported languages

Several tools provide support for only one specific language, yet some of them support more than one language. In fact, 54% of the tools support `Java`, which is the most popular supported language among the tools. Other languages well supported are `C++` 19%, `C` 18%, and `C#` 7%. Most of the 14% of the tools supporting many programming languages are commercial tools. Unfortunately, we did not find information about the languages supported in 16% of the tools evaluated. The predominance of a single specific language might not be a good indicator, since they can not be used in different domains. In fact, together with the lack of availability, their applicability and portability in the future may be at risk. On the hand, language-specific tools can be seen as an advantage so they can perform an in-depth analysis.

#### 4.3.4 RQ$_{1d}$: on the identification of self-admitted and non-admitted debt

Regarding the *non-admitted technical debt*, we found that 73% (74/101) address aspects of technical debt other than the identification itself, such as software metrics and code standards checking. This can be explained given the fact that there is a consolidated set of metrics for measuring code quality. Another explanation for the amount of tools related to code quality is that the knowledge in technical debt is still being consolidated in the software development context (Alves et al., 2016). In fact, only 16% (17/101) of the tools presented approaches to the identification of non-admitted technical debt. These tools mainly use software metrics and static source code analysis. In addition, some tools, such as JSpirit, they identify technical debt items in the project by finding *code-smells*.

Regarding the identification of *self-admitted technical debt*, we found 4% (5/101) of the identified tools propose to identify self-admitted technical debt, relying on approaches based on collecting source code comments through text mining and natural language processing techniques. With regarding to the classification of the debt found, the tools rely mostly on pattern matching such as in the MAT, using *tags* of comments (*i.e.* `TODO`, `FIXME`, `XXX`, and `HACK`) to identify debt. The small amount of tools might indicates that there is still room to further research on the identification of self-admitted technical debt. For instance, while proposing a new SATD identification tool they might consider compare the new approach with the MAT baseline Guo et al. (2019).

## 5 Empirical Evaluation

The objective of this study is the **evaluation** of *tools for the identification of technical debt items by mining source code comments*, **from the perspective** of *software engineers*, **with respect to** *the the effectiveness of the tools evaluated* taking into consideration the intersection of debt items they can find. Thus, the following research questions were defined:

**RQ$_2$:** What is the intersection between the automatic self-admitted technical debt identification approaches of the eXcomment and DebtHunter tools?

**RQ$_3$:** How are the debt items identified by the eXcomment and DebtHunter tools distributed?

**RQ$_4$:** What is the effectiveness of eXcomment and DebtHunter in identifying technical debt?

### 5.1 Evaluated Tools

We selected two tools to our evaluation study: DebtHunter (Sala et al., 2021) and eXcomment (Farias et al., 2020). The authors of this study did not participate in the development of any of the tools here evaluated. They were selected because of three reasons: *(i)* they are two out of three from the tools found in our literature review that use the same technique to identify the technical debt – source comment mining; *(ii)* they represent the *state-of-the-art* of mining source comments (out of those tools identified in our literature review); and *(iii)* by convenience, since we have already evaluated the evolution of eXcomment in another study of ours (Loiola et al., 2023). We wanted to use also a third tool is called SATDetector. However, its results output prevented us to perform the study manually as we wanted to and we did not managed to find a way around to work all three.

The DebtHunter tool uses natural language processing and machine learning. The classification is performed in two steps: *(i)* Binary classification, where comments that have or do not have self-admitted technical debt are identified, and *(ii)* multi-class classification, where comments that have debt are classified according to the type of debt they express. The tool also has two use cases, they are: *(i)* Comment marking, in this case the classification is performed based on the model pre-trained by DebtHunter and *(ii)* training a new model, in this mode the tool receives as input classified comments, which are used to train a new model. For the purpose of this case study, the tool was only used in the first use case.

The eXcomment tool uses text mining techniques to collect comments that may indicate the presence of technical debt. To this end, the tool uses a contextualized vocabulary to identify technical debt in source code comments Farias et al. (2020). The tool chooses the comments that relate to at least one pattern of the contextualized vocabulary. Finally, the patterns found in each comment are analyzed to classify them among the different types of technical debt the tool is prepared to.

**Table 7.** Target systems characterization.

| **System** (Version) | **Domain** | **LOC** | **LC** | **CVE** | **CVD** |
|---|---|---|---|---|---|
| Lottie (`v3.5.0`) | Android library | 49.441 | 2.704 | 500 | 808 |
| RocketMQ (`v4.8.0`) | Messages and Streaming | 109.261 | 22.798 | 1.203 | 2.215 |
| Trift (`v0.14.1`) | Data transportation | 311.606 | 75.494 | 1.065 | 1.350 |
| Arduino (`v1.8.16`) | Developmen environment | 149.055 | 62.643 | 1.265 | 1.202 |

**LOC:** Lines of code; **LC:** Comment lines; **CVE:** eXcomment valid comments; **CVD:** DebtHunter valid comments.

## 5.2 Target Systems

We used four target systems: RocketMQ, Lottie, Trift, and Arduino. We choose systems from different domains and sizes, open-source, and written in the programming language `Java`. The language constraint is created by the evaluated tools. Table 7 presents a brief characterization of the selected target systems, including *size in lines of code* (LOC), *amount of lines of comments* (LC), *valid comments* eXcomment (CVE) and *valid comments* DebtHunter (CVD). Valid comments indicate all comments that the tools included in their search for technical debt items during their run.

## 5.3 Oracle

We followed the scheme illustrated in Figure 2 to build our oracle. In the *first step*, the first two authors carried out the manual inspection of each comment from the source code of each target system considered independently. During this step, the evaluators identified, if any, there was a presence of technical debt in each comment, selecting only one of the types of debt presented in Table 8. In case no debt was found, the comment was classified as "without debt". In the *second step*, we manually compared both classifications from each evaluator to identify whether there was a consensus about the classification or not. In case consensus, the classification of the comment was added to the final oracle. On the other hand, in case of divergence, we resort to a third step in which we submitted the comment to a third evaluator to solve the conflict and than the comment was add to the final oracle. We calculated the Kappa coefficient to measure the agreement between the evaluators, which result in an agreement of 0.38. According to the scale proposed by Landis and Koch (1977), this score corresponds to a *fair strength of agreement*.

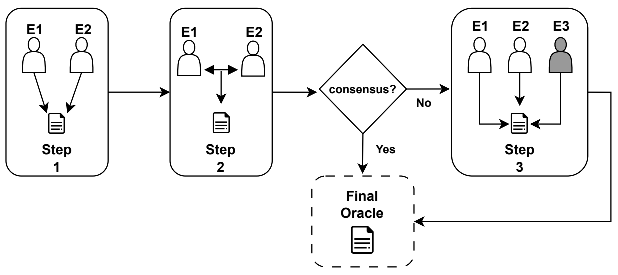


**Figure 2.** Oracle construction scheme illustrated. *(Adapted from Gomes et al. (2023))*

## 5.4 Metrics

We considered four metrics for the analysis of the results: *precision*, *recall*, $f_1$*-score*, and *accuracy*. We compared the output of the evaluated tools against the oracle built to calculation of these metrics. These metrics are defined in terms of *true positive*, *true negative*, *false positive*, and *false negative*. True positives (TP) indicate correct classifications where the tools classify the comment as technical debt, and the oracle confirms that it was indeed a debt. However, if the tools classify the comment as not being technical debt, but the oracle classifies it as technical debt, it is a *false negative* (FN). If the tools classify the comment as not containing technical debt and the oracle confirms that it did not contain technical debt, it has a *true negative* (TN). However, if the tools classify the comment as technical debt and the oracle classifies it as not being technical debt, then it is a *false positive* (FP).

The *precision* indicates the ratio between the *true positives* (TP) by the sum of the amount of *false positives* (FP) and *true positives*. In the scope of this paper, *precision* indicates the fraction of correctness in the classification of debt items when compared to the classification of the tools with the oracle. Equation 1 shows the calculation of this metric.

$$precision = \frac{TP}{TP + FP} \qquad (1)$$

The *recall* represents the number of hits of the tools in the identification of TD items. Thus, the metric represents the fraction of technical debt items effectively identified by the tools. Equation 2 shows the calculation of this metric.

$$recall = \frac{VP}{VP + FN} \quad (2)$$

The $f_1$*-score* metric is an harmonic mean between *precision* and *recall*, which represents the ratio between the number of elements and the sum of the inverses of these elements. It can vary according to the values of *precision* and *recall*. Therefore, if the values of *precision* and *recall* are high, the harmonic mean will also be high, otherwise it will be low. The $f_1$*-score* metric is calculated according to Equation 3.

$$f_1 - score = \frac{2 * precision * recall}{precision + recall} \quad (3)$$

The *accuracy* metric can be represented by the ratio of correct classifications over all classifications that have been made. In the scope of this work, *accuracy* represents the fraction of correct classifications of valid comments, whether correct for the presence or absence of technical debt. The metric is calculated according to Equation 4.

$$accuracy = \frac{TP + TN}{TP + TN + FN + FP} \quad (4)$$

## 5.5 Data Collection

We ran the eXcomment and DebtHunter tools on the target systems to collect the data we need to answer our three research questions. However, each tool identifies a different set of technical debt types. While the eXcomment (Farias et al., 2020) proposes the identification of nine different types of debt (*Documentation Debt*, *Requirements Debt*, *Defect Debt*, *Design Debt*, *Test Debt*, *Architecture Debt*, *Build Debt*, *Code Debt*, and *People Debt*), the DebtHunter (Sala et al., 2021) identifies only five (the first five enumerated for eXcomment). Therefore, we considered only the five types of debt found in both tools for this study. Table 8 defines each of them.

It is worth mentioning that, when analyzing the types of debt considered for identification by the tools, a small difference in nomenclature was noticed. What eXcomment considers *Requirements Debt*, DebtHunter considers *Implementation Debt* in the same way as the *dataset* taken as reference Maldonado and Shihab (2015) for its construction. For standardization purposes, we will only use *Requirements Debt* from this point on, as defined by Rios et al. (2018) and taken by reference in the eXcomment.

## 5.6 Syntheses

This section presents the method used to report the results regarding the research questions $RQ_2$, $RQ_3$, and $RQ_4$ obtained during the comparison between the tools.

**Table 8.** Description of the self-admitted technical debt types considered in this study.

| |
|---|
| ***Documentation Debt***: It refers to problems found in the software documentation such as absence of documentation, inadequate, or incomplete documentation. |
| ***Requirements Debt***: It refers to the activities that the development team has to implement and how the implementation will be carried out. |
| ***Defect Debt***: Refers to known defects, which the team agrees should be corrected. |
| ***Design Debt***: Refers to bad coding practices that violate the principles of object orientation. |
| ***Test Debt***: Refers to issues that may affect the quality of testing activities. |

### 5.6.1 Research Questions 2 e 3

This section presents the method used to synthesize the results obtained for research questions $RQ_2$ and $RQ_3$. The synthesis of results consisted in identifying the intersection between the approaches to identifying TD items. For this, the outputs of both tools (*i.e.*, the comments that the tools actually pointed out as evidence of TD) were compared in two steps. First, the comments classified by eXcomment were compared with the classification of DebtHunter. Second, the comments classified by DebtHunter were compared with the eXcomment classification. This form of comparison was necessary because no more efficient way of comparing the outputs manually was identified. The results obtained after comparing the outputs of the tools will be analyzed from the perspective of two distinct groups:

**Group 1:** comments that both tools pointed TD – this group will be referenced as *G1*;

**Group 2:** comments that only one of the tools pointed to TD – this group will be referred to as *G2*.

In *G1*, there were comments classified with the same types of debt by both tools (subgroup $G1_{\text{equals}}$) and also comments classified with divergence between the tools (subgroup $G1_{\text{different}}$). For example, the Lottie system contains the following comment:

```
/* These flags were in Canvas but they
were deprecated and removed. TODO:
test removing these on older versions
of Android.*/
```

In this case, while eXcomment pointed out that the comment contains *Test Debt* and *Code Debt*, DebtHunter pointed out that it contains a *Design Debt*. That is, both tools pointed out that the comment contains debt, however, of different types. Thus, the comment would be included in the group ($G1_{\text{different}}$) and would not be included in the group ($G1_{\text{equal}}$).

In *G2*, there were comments that were considered by the second tool free of technical debt ($G2_{\text{free}}$), as well as, there were comments that were considered invalid (*i.e.* not considered for classification – $G2_{\text{invalid}}$). Figure 3 shows a schema that organizes all four mentioned subgroups of *G1* ($G1_{\text{equals}}$ and $G1_{\text{different}}$) and of *G2* ($G2_{\text{free}}$ and $G2_{\text{invalid}}$).

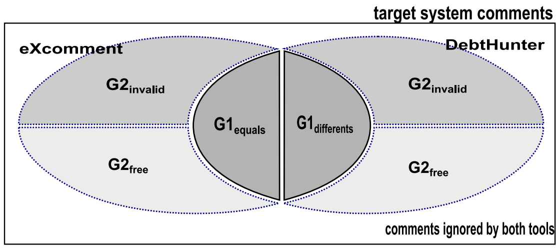


**Figure 3.** Subgroups division used on the presentation and discussion of results.

### 5.6.2 Research Questions 4

To answer RQ$_4$, the outputs of the evaluated tools were manually compared with the constructed oracle. Details of its construction were reported in Section 5.3. At the end of the comparison step, we calculated the four different metrics discussed in Section 5.4.

## 5.7 Results and Discussion

The results presented below provide an overview of the effectiveness of the two automated approaches in identifying and classifying technical debt items in software projects evaluated in this work.

### 5.7.1 RQ$_1$: on the intersection between eXcomment and DebtHunter

Table 9 presents the results of the comparison between both tools from the perspective of the two defined groups. Column "E" denotes the comparison between the classification of debt items from eXcomment with the classification from DebtHunter. Similarly, column "D" denotes the comparison between the classification of debt items from DebtHunter with the classification from eXcomment.

As results of the intersection analysis between approaches, we found that eXcomment detected a total of 233 technical debt items, this summing all debt items detected in all target systems analyzed. In turn, DebtHunter detected a total of 85 technical debt items (Table 10). Regarding the analysis of the *G1* group, we observe that DebtHunter obtained an average intersection of 7% in relation to the debt items classified by eXcomment. On the other hand, eXcomment obtained an average intersection of 19.9% in relation to the debt items classified by DebtHunter.

The result obtained in the Trift system stands out, where no cases were identified in which both tools classified as debt (*G1*). In our view, one of the factors that explain this behavior is the content of the comments found in Trift. For example, in 14 of the 15 *Requirements Debt* found by DebtHunter went to the comment "`TODO(mcslee): implement`". However, as eXcomment uses textual standards for debt identification Farias et al. (2020), the standards defined by the tool's heuristics do not recognize debt items in a comment containing a single word ("implement").

As for the analysis of the $G2_{\text{free}}$ group, of the debt items pointed out by eXcomment, on average, 79.3% of them DebtHunter classified as "technical debt free". Similarly, of the debt items indicated by DebtHunter, 27.1% of them eXcomment classified as "technical debt free". Finally, as for the subgroup $G2_{\text{invalid}}$, of the debt items classified by eXcomment, 13.7% of them DebtHunter considered invalid. On the other hand, of the items classified as debt by DebtHunter, 63.6% were considered invalid by eXcomment.

**Table 9.** Comparative analysis of the intersection of results between DebtHunter e eXcomment.

| Group | Lottie | | RocketMQ | | Trift | | Arduino | |
|---|---|---|---|---|---|---|---|---|
| | E | D | E | D | E | D | E | D |
| ***G1*** | 2 | 2 | 2 | 2 | 0 | 0 | 18 | 18 |
| – $G1_{\text{differents}}$ | 2 | 2 | 1 | 1 | 0 | 0 | 12 | 12 |
| – $G1_{\text{equals}}$ | 0 | 0 | 1 | 1 | 0 | 0 | 6 | 6 |
| ***G2*** | 36 | 8 | 30 | 15 | 53 | 19 | 92 | 32 |
| – $G2_{\text{free}}$ | 31 | 3 | 26 | 0 | 40 | 5 | 87 | 20 |
| – $G2_{\text{invalid}}$ | 5 | 5 | 4 | 15 | 13 | 15 | 5 | 12 |

**E:** comparison of the classification of debt items by eXcomment with regarding to DebtHunter; **D** comparison of the classification of debt items by DebtHunter with regarding to eXcomment.

After analyzing the debt items classified by the tools, we found that eXcomment has a higher percentage of intersection in relation to the debt items identified by DebtHunter. On average, out of the debt items identified by eXcomment, DebtHunter was able to identify only 7% of them ($G1_{\text{total}}$–E). On the other hand, out of the debt items identified by DebtHunter, the eXcomment tool was able to identify 19.9% of them ($G1_{\text{total}}$–D).

In general, the evaluation results showed that eXcomment tends to be more sensitive in identifying debt items than DebtHunter. On average, DebtHunter considered that there was no debt ($G2_{\text{free}}$ – E) in 79.3% of the comments containing debt according to eXcomment. On the other hand, on average, eXcomment considered that there was no debt ($G2_{\text{free}}$ – D) in only 27.5% of the comments containing debt according to DebtHunter.

We also noticed that DebtHunter tends to consider fewer invalid comments compared to eXcomment. On average, DebtHunter considered invalid ($G2_{\text{invalid}}$ – E) only 13.7% of the comments that eXcomment considered valid. On the other hand, on average, eXcomment considered 63.6% of the comments considered valid by DebtHunter as invalid ($G2_{\text{invalid}}$ – D).

**In summary, the intersection of debt items classified by the eXcomment and DebtHunter tools is small (less than 20%) in relation to the number of items pointed out by each of them.**

### 5.7.2 RQ$_2$: on the debt items distribution

Table 10 presents the type of debt and the amount of technical debt items identified by each tool in the target systems. The column "type of debt" shows the type of debt that the items represent, while the column "T" indicates which tool this debt occurred, and the column "TOTAL" indicates the total debt by type in each tool. The following columns indicate in which system the technical debt items occurred. Finally, the *row* “TOTAL” indicates the total of technical debt items identified in each tool.

Analyzing the distribution of the types of debt identified by both tools, there were few or no items of the types *Documentation Debt* and *Test Debt*. While only eXcomment found *Test Debt*, neither tool identified *Documentation Debt* items. On the other hand, both tools found several items of *Design Debt*, *Defect Debt* and *Requirements Debt*. While eXcomment found more items of *Defect Debt*, *Design Debt*, and *Requirements Debt* (descending order), DebtHunter found more items of *Design Debt*, *Requirements Debt*, and *Defect Debt* (descending order).

Analyzing these results in perspective of the size of the target systems, it was observed that eXcomment identifies a higher number of technical debt items per line of code than DebtHunter. On average, while eXcomment identified 4.9 debt items per 10K lines of code, DebtHunter identified only 1.68 items. On the other hand, regarding the number of valid comments of each system, we also found that eXcomment identified a higher number of debt items than DebtHunter. On average, while eXcomment identified 5.9 technical debt items per 100 valid comments, DebtHunter identified only 1.2 items.

**In summary, the distribution of debts found by the eXcomment and DebtHunter tools are concentrated in *Design Debt*, *Defect Debt*, and *Requirements Debt***

.

**Table 10.** Quantity of debt items identified by each tool in each target system.

| Type of Debt | T | Lottie | RocketMQ | Trift | Arduino | TOTAL |
|---|---|---|---|---|---|---|
| *Design Debt* | EX | 6 | 3 | 6 | 14 | 29 |
| | DH | 5 | 16 | 4 | 30 | 55 |
| *Defect Debt* | EX | 15 | 19 | 27 | 51 | 112 |
| | DH | 0 | 0 | 1 | 3 | 4 |
| *Requirements Debt* | EX | 5 | 2 | 2 | 8 | 17 |
| | DH | 4 | 0 | 15 | 7 | 26 |
| *Documentation Debt* | EX | 0 | 0 | 0 | 0 | 0 |
| | DH | 0 | 0 | 0 | 0 | 0 |
| *Test Debt* | EX | 1 | 0 | 0 | 0 | 1 |
| | DH | 0 | 0 | 0 | 0 | 0 |
| TOTAL | EX* | 38 | 32 | 53 | 110 | 233 |
| | DH | 9 | 16 | 20 | 42 | 85 |

**EX**: eXcomment; **DH**: DebtHunter; **T**: Tools; *****: The existing difference in the total o debt items identified by eXcomment in each system is explained by the items of a debt types DebtHunter can not identify.

### 5.7.3 RQ$_3$: on the effectiveness of eXcomment and DebtHunter on the identification of technical debt

***Metrics***: Tables 11, 12, 13 show the results of both tools eXcomment and DebtHunter in detecting technical debt items in the target systems analyzed according to the proposed metrics. The columns present, respectively, the types of debt identified, the tool that identified the debt, the number of debt items identified in the oracle($Q_i$), and the results, in percentage, of the metrics of interest. Debt types that were not found in the oracle were represented with “$-$” since the calculation of their metrics would not be possible. In addition, the analysis of the metrics of the RocketMQ system could not be conducted due to the scarcity of debts identified by the tools, which made it impossible to calculate the metrics of interest.

*Precision* and *recall* depend directly on the amount of TP, which results in zero percent of both metrics in some types of debt, since the systems identified few or no TP. Similarly, the $f_1$-*score* is known to depend directly on the *precision* and *recall*. The fact that both metrics were set to zero in some cases influences the result of the $f_1$-*score*. Finally, the *accuracy* corresponds to the total number of correct predictions of technical debt items in relation to the total number of predictions made by the tools. Therefore, it depends directly on the TP and TN values. As the systems identified few or no TPs, the number of TNs is almost as higher as the number of valid comments. Thus, it resulted 100% accuracy for some cases based only on the number of TN, which is calculated through the difference between the number of valid comments and the number of debt classifications made by the tools. For example, in the case of *Test Debt* for the Lottie, eXcomment identified only one FN resulting in 499 TN. Thus, the *accuracy* would be calculated by the quotient of the amount of TN by all the quantities expected in the system (*i.e.* TP, FP, FN and TN), resulting, inaccurately, in an *accuracy* of 100%.

**Table 11.** Results eXcomment and DebtHunter in the identification of debt items on Lottie *(in %)*.

| Type of Debt | $Q_i$ | T | *P* | *C* | *F$_1$* | *A* |
|---|---|---|---|---|---|---|
| *Requirements Debt* | 1 | EX | 0 | 0 | 0 | 99 |
| | | DH | 0 | 0 | 0 | 100 |
| *Defect Debt* | 3 | EX | 7 | 33 | 11 | 97 |
| | | DH | 0 | 0 | 0 | 100 |
| *Design Debt* | 0 | EX | – | – | – | – |
| | | DH | – | – | – | – |
| *Test Debt* | 1 | EX | 0 | 0 | 0 | 100 |
| | | DH | 0 | 0 | 0 | 100 |
| *Documentation Debt* | 0 | EX | – | – | – | – |
| | | DH | – | – | – | – |

**EX**: eXcomment; **DH**: DebtHunter; **T**: Tool; **Q$_i$**: Quantity of debt items; **P**: *precision*; **C**:*recall*; **F$_1$**: $f_1$-*score*; **A**: *accuracy*

***Recurrent Debts***: The results show eXcomment to be more likely to identify *Defect Debt* compared to the DebtHunter. Besides, both of them were considerably imprecise in all classification. Considering all the target systems analyzed, eX-

**Table 12.** Results eXcomment and DebtHunter in the identification of debt items on Trift *(in %)*.

| Type of Debt | $Q_i$ | T | *P* | *C* | $F_1$ | *A* |
|---|---|---|---|---|---|---|
| *Requirements Debt* | 2 | EX | 50 | 100 | 66 | 100 |
| | | DH | 0 | 0 | 0 | 100 |
| *Defect Debt* | 4 | EX | 10 | 66 | 17 | 98 |
| | | DH | 0 | 0 | 0 | 100 |
| *Design Debt* | 1 | EX | 0 | 0 | 0 | 100 |
| | | DH | 0 | 0 | 0 | 100 |
| *Test Debt* | 1 | EX | 0 | 0 | 0 | 100 |
| | | DH | 0 | 0 | 0 | 100 |
| *Documentation Debt* | 4 | EX | 0 | 0 | 0 | 100 |
| | | DH | 0 | 0 | 0 | 100 |

**EX**: eXcomment; **DH**: DebtHunter; **T**: Tool; $\mathbf{Q}_i$: Quantity of debt items; **P**: *precision*; **C**:*recall*; $\mathbf{F}_1$:*$f_1$-score*; **A**: *accuracy*

**Table 13.** Results eXcomment and DebtHunter in the identification of debt items on Arduino *(in %)*.

| Type of Debt | $Q_i$ | T | *P* | *C* | $F_1$ | *A* |
|---|---|---|---|---|---|---|
| *Requirements Debt* | 5 | EX | 0 | 0 | 0 | 99 |
| | | DH | 0 | 0 | 0 | 100 |
| *Defect Debt* | 13 | EX | 12 | 30 | 17 | 97 |
| | | DH | 100 | 7 | 14 | 99 |
| *Design Debt* | 8 | EX | 0 | 0 | 0 | 99 |
| | | DH | 50 | 12 | 20 | 99 |
| *Test Debt* | 0 | EX | – | – | – | – |
| | | DH | – | – | – | – |
| *Documentation Debt* | 3 | EX | 0 | 0 | 0 | 100 |
| | | DH | 0 | 0 | 0 | 100 |

**EX**: eXcomment; **DH**: DebtHunter; **T**: Tool; $\mathbf{Q}_i$: Quantity of debt items; **P**: *precision*; **C**:*recall*; $\mathbf{F}_1$:*$f_1$-score*; **A**: *accuracy*

comment presented an average of only 7% of *precision* in the detection of *Defect Debt* debt. Yet, eXcomment achieved only 32% of coverage on average in the identification of *Defect Debt* debts. This fact, indicates that despite marking many *Defect Debt* debts, it is not very assertive in identifying this type of debt. In addition, eXcomment obtained, on average, a *$f_1$-score* of 11% and an accuracy of 97% in relation to the identification of default debts.

It is worth mentioning that, DebtHunter identified technical debt only in Arduino. It identified both *Defect Debt* and *Design Debt*. While identifying *Defect Debt*, DebtHunter reached *precision* of 100%. However, as DebtHunter scored only a single TP and no FPs such *precision* results is misleading. It only reached 7% of *recall* and 14% of *$f_1$-score*. Finally, DebtHunter reached 99% *accuracy* in the identification of both *Defect Debt* and *Design Debt*. Moreover, results show it reached 50% of *precision*, 12% of *recall*, and 20% of *$f_1$-score* in identifying *Design Debt*.

***Debts in relation to valid comments and lines of code*:** we observed that, considering only correct classifications, both tools identified similar amount of debt items with regarding to the valid comments. On average, eXcomment identifies 1.16 debt items for every 100 valid comments, while DebtHunter identifies only 1 item. Moreover, when taking into account the size in lines of code of the target systems, we identified, on average, 0.90 items every 10K lines of code, covering only the types of debts considered for this study.

***Analysis of errors*:** Figure 4 presents a set of *Sankey diagrams* showing a mapping from the wrong classification of comments to the expected classification according to the oracle. We included an interactive version of such diagrams in the replication package Gomes et al. (2024). While Figures 4a, 4c, 4e, and 4g map the errors of eXcomment, the Figures 4b, 4d, 4f, and 4h map the errors of DebtHunter. More specifically, while the tools′ classifications are located on the left side of the *Sankey diagram*, the oracle classifications are on the right side. Additionally, these diagrams use arrows of different widths and colors to illustrate where and how the tools mistakenly classified the systems valid comments according to the oracle.

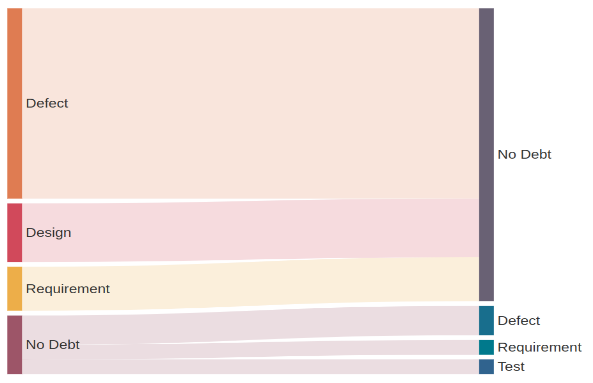
We decide to include such analysis because of the amount of *false positives* and *false negatives*, as it can be seen in the figures. The diagrams allow us to intuitively identify patterns of errors in the identification of technical debt. In fact, we can understand from the diagrams that while eXcomment incur in a significant number of false positives while classifying *Defect Debt*, DebtHunter incur in a significant number of false positives while classifying *Design Debt*. Additionally, it can be seen that DebtHunter incur relatively more false negatives, as we noticed due to the lack of true positives in most target systems. Overall, the errors of the analyzed tools differ significantly, which suggest that both approaches need further research to improve how they to work in an acceptable manner.

**In summary, the tools demonstrated some ability to identify *Defect Debt* and *Design Debt*, but with low *precision* and *recall*. Moreover, they had great difficulty in identifying the other types of debt under evaluation, namely *Documentation Debt*, *Test Debt* and *Requirements Debt*.**

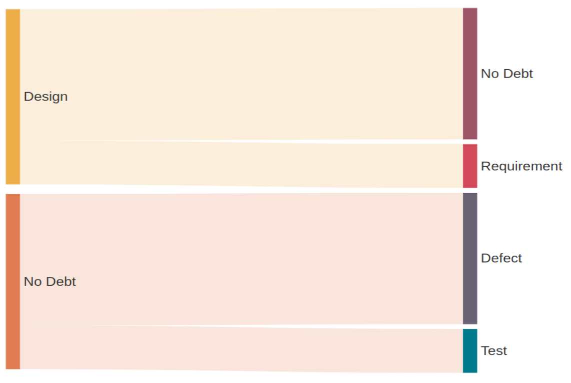
## 6 Threats to Validity

In any research study, threats to validity are common. In the following, we discuss some of the threats of this study, as well as the steps taken to bypass or minimize their effects.

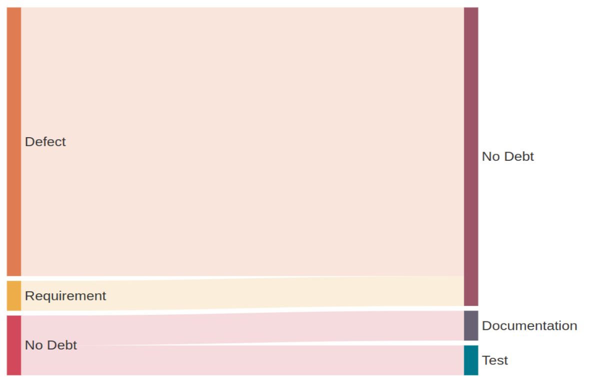
**Conclusion validity:** it is concerned with the relationship between treatment and outcome. Unfortunately, both the number of target systems and the amount of debt found manually in each system is not sufficient for a general conclusion. However, we sought to evaluate them carefully with impartiality to reach best conclusion relying only on the gathered data and well known metrics. Additionally, to minimize the fatigue on the manual inspection of the comments to build our oracle, we relied on two independent evaluation, as well as third one to eventually resolve divergence on the manual classification.



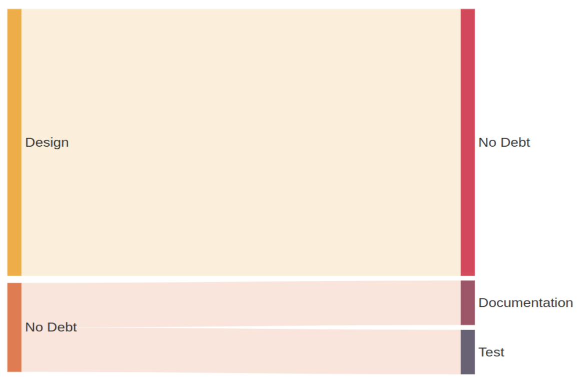
**(a)** eXcomment – Lottie



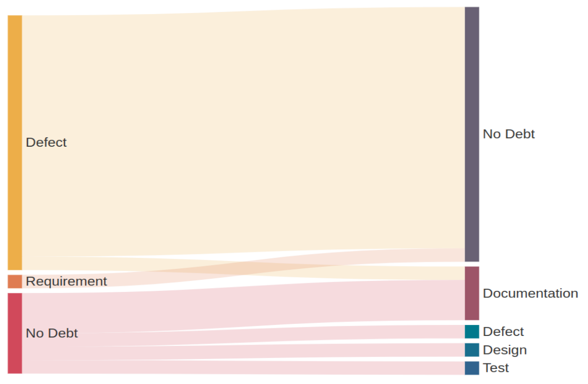
**(b)** DebtHunter – Lottie



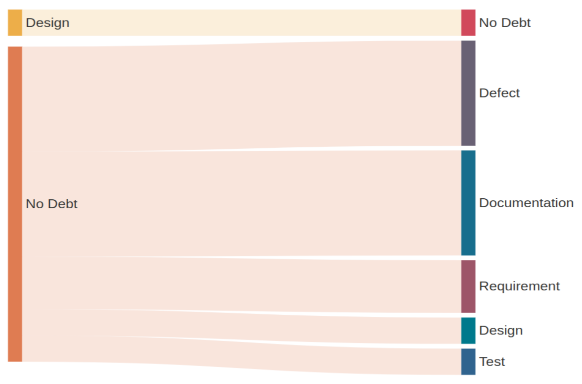
**(c)** eXcomment – RocketMQ



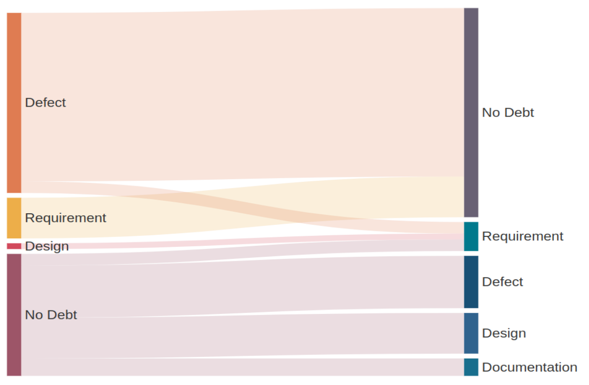
**(d)** DebtHunter – RocketMQ



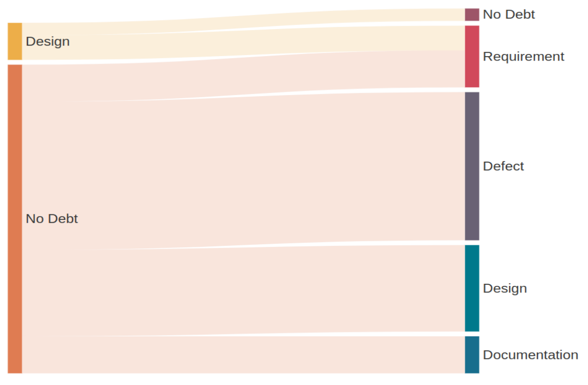
**(e)** eXcomment– Trift



**(f)** DebtHunter – Trift



**(g)** eXcomment – Arduino



**(h)** DebtHunter – Arduino

**Figure 4.** Sankey diagrams of tool errors against the oracle on each target system.

**Internal validity:** it is concerned whether there is a causal relationship between treatment and outcome. In this case, we selected open source target system from different domains to avoid and tools we did not developed to improve our internal validity.

**Construct validity:** It refers to the extent to which the experiment setting reflects the theory. To address the construction validity we selected systems from different sizes so it could better represent other software. Additionally, one of the researchers working on the oracle construction is undergraduate and he has little experience on the software development. We minimize this threat by conducting independent evaluations by researcher of different levels of experience. Another point of threat is the extension of the literature review of tools. We selected papers only by scanning and not an full reading since there is several systematic literature reviews on the topic. Additionally, we could not access 4 references for the full reading and extraction of possible information. Such fact may cause our results to be incomplete or imprecise. However, we tried our best to get all tools we could find during the review.

**External validity:** All for target systems small of medium-sized systems. Thus, there is no guarantee that the results presented by them can be generalized. Additionally, the observations presented cannot be generalized to other development contexts, nor to other software domains.

## 7 Conclusion and Future Work

In this study, we reviewed technical debt tools in the literature with regard to their the phase they can be used in he management of the technical debt life-cycle, their availability, the programming languages supported and the type of technical debt they focus. The review yielded a set of tools available in the literature, from which we selected two of them to perform an empirical evaluation. The evaluation took into consideration *(i)* the intersection of their classifications, *(ii)* the distribution of technical debt items identified by both of them, and *(iii)* the effectiveness of each of them. Our results raised evidence of the effectiveness of the eXcomment and DebtHunter tools in identifying and classifying technical debt as well as their weaknesses.

As the empirical evaluation in this paper extends a previous one, we now integrate the findings of both studies. In the original study (Sections 5.7.1 and 5.7.2), while results showed that DebtHunter debt classification overlaps the classification of the eXcomment classification by 7%, they also showed that the eXcomment classification overlaps the DebtHunter classification by 19.9%. Additionally, despite the disagreement between eXcomment and DebtHunter regarding the amount of TD items present in each of the target systems, the tools seem to converge regarding the types *Design Debt*, *Defect Debt* and *Requirements Debt*. In the extension study (Section 5.7.3), eXcomment and DebtHunter mostly identified *Defect Debt* and *Design Debt* in the target systems. On top of that, we found they have low precision and recall on the identification of these types of debt.

Although both tools use state-of-the-art technologies to extract and classify the content of the source code comments, they still could not recognize the different types of debt with high level of reliability. We believe that use the code comments as the only source of evidence to all types of debt might be a problem due to the nature of each type and the amount of useful information in the comments itself. In fact, our Sankey diagram analysis found different pattern of errors in each tool. While eXcomment incur more frequently in false positives, DebtHunter incur more frequently in false negatives. Moreover, we found that DebtHunter were less effective in identifying technical debt when compared to eXcomment. Neither of the tools evaluated managed to identify *Requirements Debt*, *Test Debt*, and *Documentation Debt*. These facts highlight the need for improvements in both approaches to increase *precision* and *recall* on the the debt items identification, as well as different types of debt.

As future work, it is possible to investigate the behavior of the tools in more target systems of other domains and sizes to build more sound evidence for our conclusions and raise evidence for the differences in pattern of error. In addition, it is possible to evaluate other tools available in the literature with the same purpose, as well as to carry out the construction of a more robust oracle, using more systems and classified by experts in Technical Debt.

## References

Alves, N. S., Mendes, T. S., de Mendonça, M. G., Spínola, R. O., Shull, F., and Seaman, C. (2016). Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121.

Ampatzoglou, A., Chatzigeorgiou, A., Arvanitou, E. M., and Bibi, S. (2022). Sdk4ed: A platform for technical debt management. *Software: Practice and Experience*, 52(8):1879–1902.

Avgeriou, P. C., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., Saarimaki, N., Sas, D. D., de Toledo, S. S., and Tsintzira, A. A. (2021). An overview and comparison of technical debt measurement tools. *IEEE Software*, 38(3):61–71.

Bavota, G. and Russo, B. (2016). A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 315–326, New York, NY, USA. Association for Computing Machinery.

Chandramouli, P., Codabux, Z., and Vidoni, M. (2022). analyzer: A sonarqube plugin for analyzing object-oriented r packages. *SoftwareX*, 19:101113.

Cunningham, W. (1992). The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30.

Farias, M., Mendes, T. S., Mendonça, M. G., and Spínola, R. O. (2021). On comment patterns that are good indicators of the presence of self-admitted technical debt and those that lead to false positive items. In *Proceedings of the 27th Annual Americas Conference on Information Systems (AMCIS)*, pages 1–10.

Farias, M. A. F., de Mendonça Neto, M. G., Kalinowski, M., and Spínola, R. O. (2020). Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information Software Technologies*, 121:106270.

Fontana, F. A., Roveda, R., and Zanoni, M. (2016). Technical debt indexes provided by tools: a preliminary discussion. In *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, pages 28–31. IEEE.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison Wesley Professional, United States.

Freire, S., Rios, N., Mendonça, M., Falessi, D., Seaman, C., Izurieta, C., and Spínola, R. O. (2020). Actions and impediments for technical debt prevention: results from a global family of industrial surveys. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1548–1555.

Freitas, G., Reboucas, R., and Coelho, R. (2022). Exploring technical debt tools: A systematic mapping study. In *Enterprise Information Systems: 23rd International Conference, ICEIS 2021, Virtual Event, April 26–28, 2021, Revised Selected Papers*, volume 455, page 280. Springer Nature.

Gomes, F., Santos, E., Freire, S., Mendes, T. S., Mendonça, M., and Spínola, R. (2023). Investigating the point of view of project management practitioners on technical debt - a study on stack exchange. *Journal of Software Engineering Research and Development*, 11(1):12:1 – 12:15.

Gomes, F. G. S., Mendes, T. S., Spínola, R. O., Mendonça, M., and Farias, M. (2019). Uma análise da relação entre code smells e dívida técnica auto-admitida. In *Anais do VII Workshop on Software Visualization, Evolution and Maintenance (VEM)*, pages 37–44, Porto Alegre, RS, Brasil. SBC.

Gomes, T., Loiola, D., and Santos, A. R. (2022). Avaliação de ferramentas de identificação de dívida técnica auto-admitida. In *Anais do X Workshop de Visualização, Evolução e Manutenção de Software*, pages 16–20, Porto Alegre, RS, Brasil. SBC.

Gomes, T., Loiola, D., and Santos, A. R. (2024). Technical debt tools: a survey and an empirical evaluation. Online at `http://www.doi.org/10.5281/zenodo.8198082`. Creative Commons 4.0 Licensed Dataset.

Guo, Z., Liu, S., Liu, J., Li, Y., Chen, L., Lu, H., Zhou, Y., and Xu, B. (2019). Mat: A simple yet strong baseline for identifying self-admitted technical debt.

Landis, J. R. and Koch, G. G. (1977). The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

Lefever, J., Cai, Y., Cervantes, H., Kazman, R., and Fang, H. (2021). On the lack of consensus among technical debt detection tools. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130. IEEE.

Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., and Fontana, F. A. (2021a). A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software*, 171:110827.

Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., and Fontana, F. A. (2021b). A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software*, 171:110827.

Lenarduzzi, V., Pecorelli, F., Saarimaki, N., Lujan, S., and Palomba, F. (2022). A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software*, 198:111575.

Lenarduzzi, V., Sillitti, A., and Taibi, D. (2020). A survey on code analysis tools for software maintenance prediction. In *Proceedings of 6th International Conference in Software Engineering for Defence Applications: SEDA 2018 6*, pages 165–175. Springer.

Li, Y., Soliman, M., and Avgeriou, P. (2023). Automatic identification of self-admitted technical debt from four different sources. *Empirical Software Engineering*, 28(3):1–38.

Li, Z., Avgeriou, P., and Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220.

Loiola, D. A. M., Gomes, T. B. S., Santos, A. R., and Farias, M. A. F. (2023). Avaliação do uso de heurísticas de contexto na mineração de dívida técnica com a eXComment. *Revista de Sistemas e Computação.*, 13.

Maldonado, E. D. S. and Shihab, E. (2015). Detecting and quantifying different types of self-admitted technical debt. In *Proceedings of the 7th Workshop on Managing Technical Debt*, MTD '11, pages 9–15, New York, NY, USA. IEEE.

Mortara, J., Bavota, G., Lanza, M., and Panichella, S. (2022). Customizable visualization of quality metrics for object-oriented variability implementations. In *26th ACM International Systems and Software Product Line Conference (SPLC)*, pages 1–12. ACM.

Murillo, M. I., López, G., Spínola, R., Guzmán, J., Rios, N., and Pacheco, A. (2023). Identification and management of technical debt: A systematic mapping study update. *Journal of Software Engineering Research and Development*, 11(1):8:1 – 8:20.

Nikolaidis, N., Ampatzoglou, A., Chatzigeorgiou, A., Tsekeridou, S., and Piperidis, A. (2022). Technical debt in service-oriented software systems. In *International Conference on Product-Focused Software Process Improvement*, pages 265–281. Springer.

Oliveira, I., Marques-Neto, H., and Xavier, L. (2020). Analisando estratégias para identificação de dívidas técnicas. In *Anais do VIII Workshop de Visualização, Evolução e Manutenção de Software*, pages 9–16, Porto Alegre, RS, Brasil. SBC.

Potdar, A. and Shihab, E. (2014). An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 91–100, Victoria, BC, Canada. IEEE.

Rios, N., de Mendonça Neto, M. G., and Spínola, R. O. (2018). A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology*, 102:117–145.

Sala, I., Tommasel, A., and Arcelli Fontana, F. (2021). Debthunter: A machine learning-based approach for detecting self-admitted technical debt. In *Evaluation and Assessment in Software Engineering*, EASE 2021, page 278–283, New York, NY, USA. Association for Computing Machinery.

Tom, E., Aurum, A., and Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516.

Tsoukalas, D., Chatzigeorgiou, A., Ampatzoglou, A., Mittas, N., and Kehagias, D. (2022). Td classifier: Automatic identification of java classes with high technical debt. In *Proceedings of the International Conference on Technical Debt*, pages 76–80. ACM.

Venters, C. C., Capilla, R., Betz, S., Penzenstadler, B., Crick, T., Crouch, S., Nakagawa, E. Y., Becker, C., and Carrillo, C. (2018). Software sustainability: Research and practice from a software architecture viewpoint. *Journal of Systems and Software*, 138:174–188.